QT

Team Emertxe

Painting & styling

Objectives

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Objectives

- Painting
  - You paint with a painter on a paint device during a paint event
  - Qt widgets know how to paint themselves
  - Often widgets look like we want
  - Painting allows device independent 2D visualization
  - Allows to draw pie charts, line charts and many more
- StyleSheets
  - Fine grained control over the look and feel
  - Easily applied using style sheets in CSS format

Module Objectives

Covers techniques for general 2D graphics and styling applications.

- **Painting**
  - Painting infrastructure
  - Painting on widget
- **Color Handling**
  - Define and use colors
  - Pens, Brushes, Palettes
- **Shapes**
  - Drawing shapes
- **Transformation**
  - 2D transformations of a coordinate system
- **Style Sheets**
  - How to make small customizations
  - How to apply a theme to a widget or application

Painting & Styling

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

QPainter

- Paints on paint devices (QPaintDevice)
- QPaintDevice implemented by
  - On-Screen: QWidget
  - Off-Screen: QImage, QPixmap
  - And others …
- Provides drawing functions
  - Lines, shapes, text or pixmaps
- Controls
  - Rendering quality
  - Clipping
  - Composition modes

Painting
on Widgets

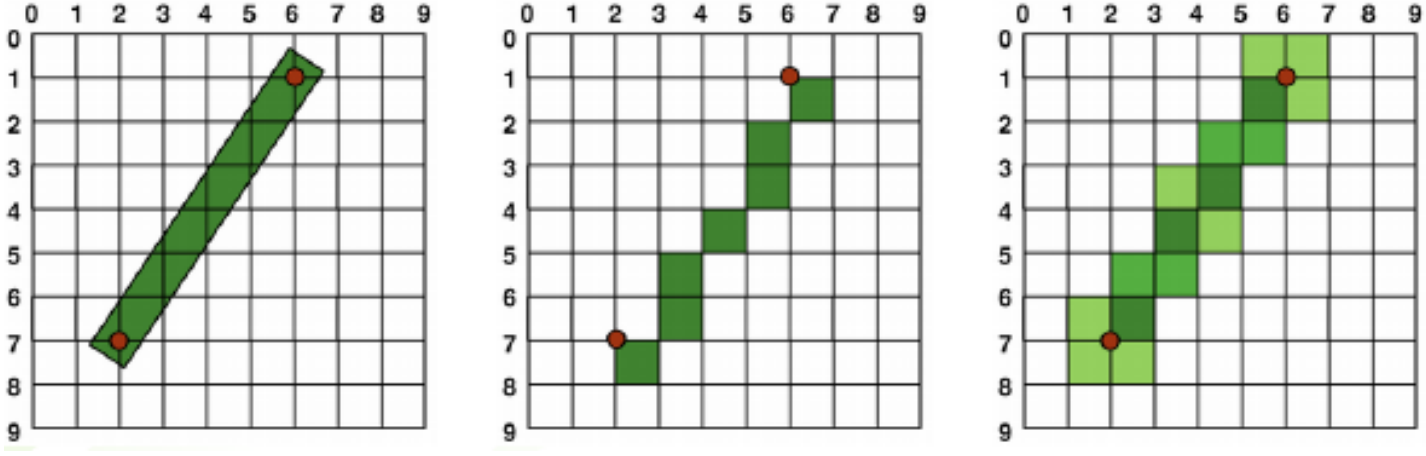
- Override paintEvent(QPaintEvent*)

```
void CustomWidget::paintEvent(QPaintEvent *)
{
  QPainter painter(this);
  painter.drawRect(0,0,100,200); // x,y,w,h
}
```

- Schedule painting
  - `update()`: schedules paint event
  - `repaint()`: repaints directly
- Qt handles double-buffering
- To enable filling background:
  - `QWidget::setAutoFillBackground(true)`

Coordinate System

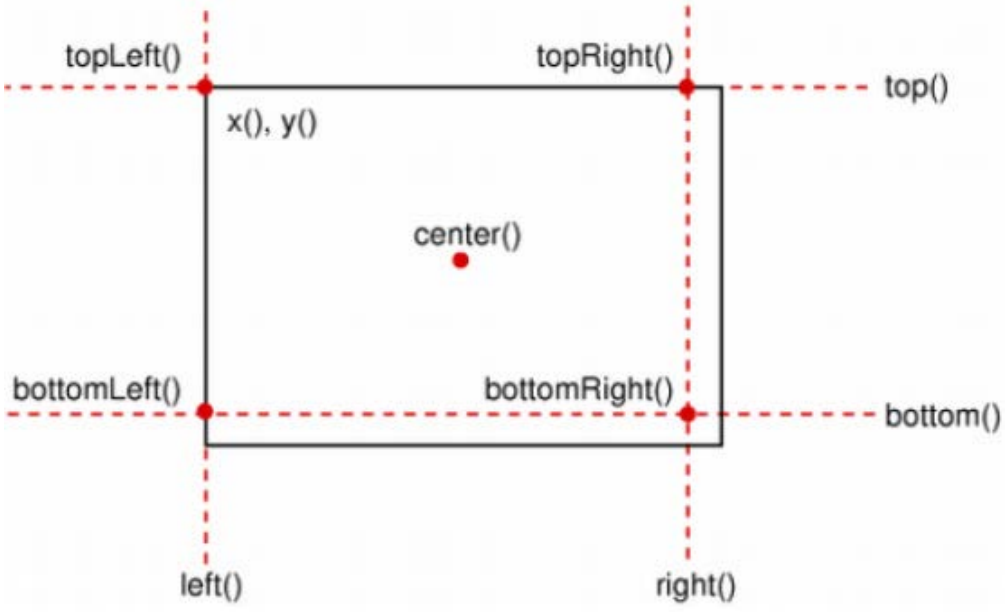
- Controlled by QPainter
- Origin: Top-Left
- Rendering
  - Logical - mathematical
  - Aliased - right and below
  - Anti-aliased – smoothing



- Rendering quality switch
- QPainter::setRenderHint()

Geometry Classes

- QSize(w,h)
  - scale, transpose
- QPoint(x,y)
- QLine(point1, point2)
  - translate, dx, dy
- QRect(point, size)
  - adjust, move
  - translate, scale, center




```
QSize size(100,100);
QPoint point(0,0);
QRect rect(point, size);
rect.adjust(10,10,-10,-10);
QPoint center = rect.center();
```

Painting & Styling

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Color Values

- Using different color models:
  - `QColor(255,0,0) // RGB`
  - `QColor::fromHsv(h,s,v) // HSV`
  - `QColor::fromCmyk(c,m,y,k) // CMYK`
- Defining colors:

```
QColor(255,0,0); // red in RGB
QColor(255,0,0, 63); // red 25% opaque (75% transparent)
QColor("#FF0000"); // red in web-notation
QColor("red"); // by svg-name
Qt::red; // predefined Qt global colors
```

- Many powerful helpers for manipulating colors

```
QColor("black").lighter(150); // a shade of gray
```

- QColor always refers to device color space

QPen

- A pen (QPen) consists of:
  - a color or brush
  - a width
  - a style (e.g. NoPen or SolidLine)
  - a cap style (i.e. line endings)
  - a join style (connection of lines)
- Activate with QPainter::setPen().

```
QPainter painter(this);
QPen pen = painter.pen();
pen.setBrush(Qt::red);
pen.setWidth(3);
painter.setPen(pen);
// draw a rectangle with 3 pixel width red outline
painter.drawRect(0,0,100,100);
```

The Outline

Rule

The outline equals the size plus half the pen width on each side.

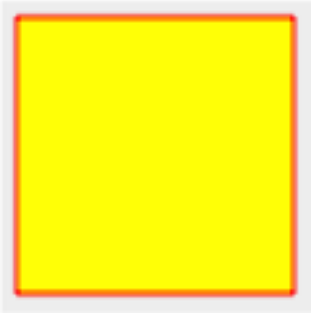
- For a pen of width 1:

```
QPen pen(Qt::red, 1); // width = 1
float hpw = pen.widthF()/2; // half-pen width
QRectF rect(x,y,width,height);
QRectF outline = rect.adjusted(-hpw, -hpw, hpw, hpw);
```

- *Due to integer rounding on a non-antialiased grid, the outline is shifted by 0.5 pixel towards the bottom right.*
- Demo

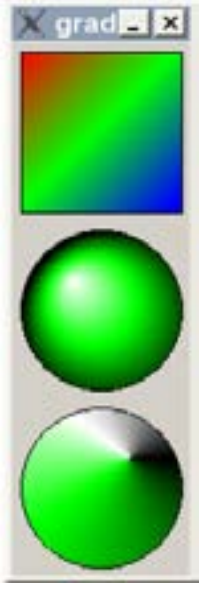
QBrush

- QBrush defines fill pattern of shapes
- Brush configuration
  - setColor(color)
  - setStyle(Qt::BrushStyle)
    - NoBrush, SolidPattern, ...
  - QBrush(gradient) // QGradient's
  - setTexture(pixmap)
- Brush with solid red fill



```
painter.setPen(Qt::red);
painter.setBrush(QBrush(Qt::yellow,
Qt::SolidPattern));
painter.drawRect(rect);
```

Gradient fills



- Gradients used with QBrush
- Gradient types
  - QLinearGradient
  - QConicalGradient
  - QRadialGradient
- Gradient from P1(0,0) to P2(100,100)

```
QLinearGradient gradient(0, 0, 100, 100);
// position, color: position from 0..1
gradient.setColorAt(0, Qt::red);
gradient.setColorAt(0.5, Qt::green);
gradient.setColorAt(1, Qt::blue);
painter.setBrush(gradient);
// draws rectangle, filled with brush
painter.drawRect(0, 0, 100, 100 );
```

- Demo

ΣMERTXE

Brush on QPen

- Possible to set a brush on a pen
- Strokes generated will be filled with the brush

- Demo

Color Themes
and Palettes

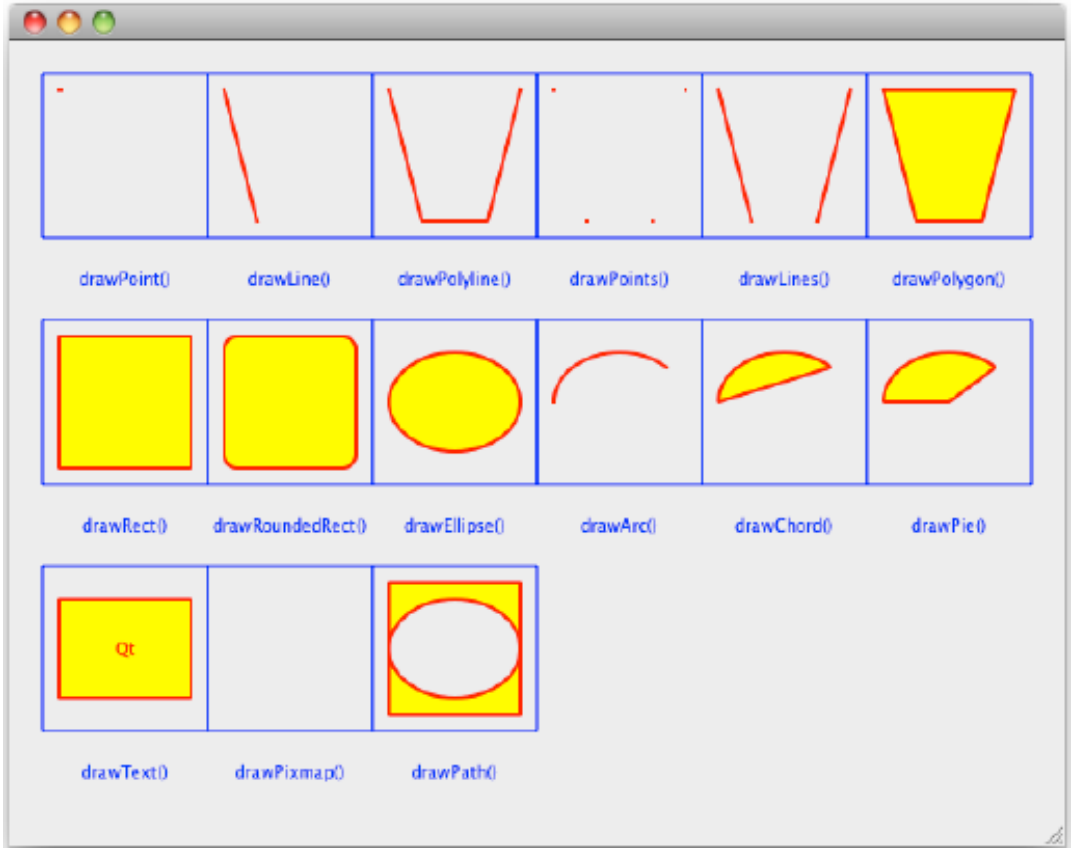
- To support widgets color theming
  - `setColor(blue)` not recommended
  - Colors needs to be managed
- QPalette manages colors
  - Consist of color groups
- `enum QPalette::ColorGroup`
  - Resemble widget states
    - `QPalette::Active`
  - Used for window with keyboard focus
    - `QPalette::Inactive`
  - Used for other windows
    - `QPalette::Disabled`
  - Used for disabled widgets

Painting & Styling

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Drawing Figures

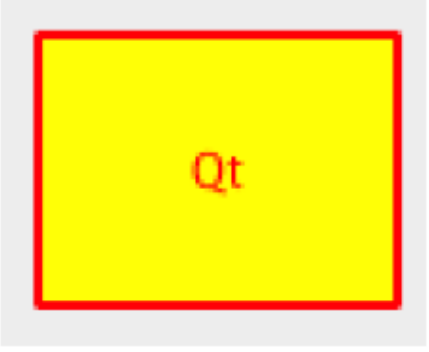
- Painter configuration
  - pen width: 2
  - pen color: red
  - font size: 10
  - brush color: yellow
  - brush style: solid
- Demo



Drawing Text

- **QPainter::drawText(rect, flags, text)**

```
QPainter painter(this);
painter.drawText(rect, Qt::AlignCenter,
tr("Qt"));
painter.drawRect(rect);
```

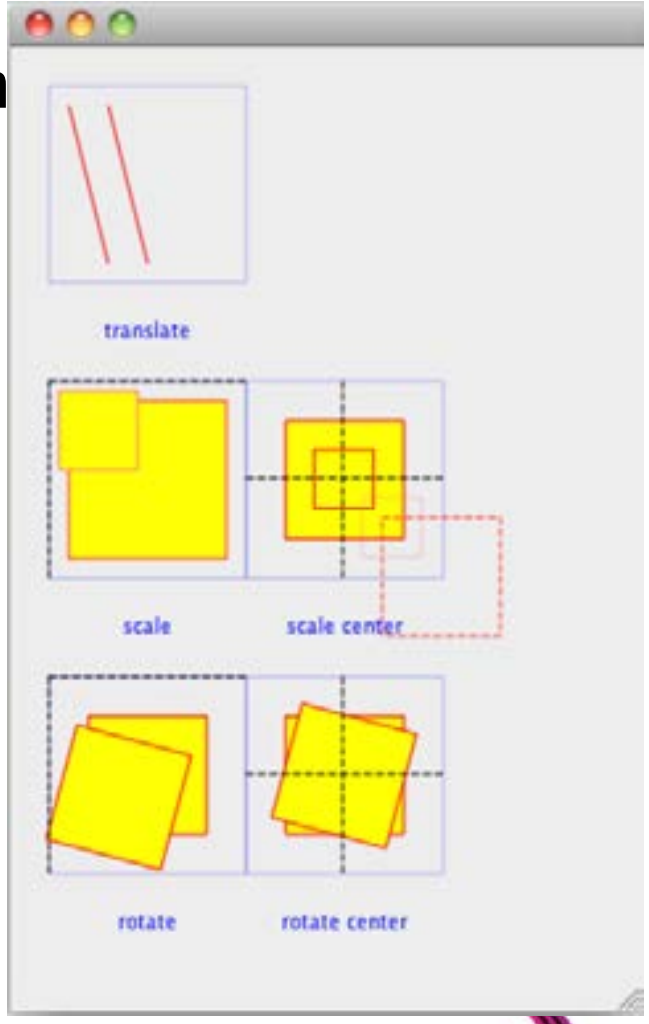


- QFontMetrics
  - calculate size of strings

```
QFont font("times", 24);
QFontMetrics fm(font);
int pixelsWide = fm.width("Width of this text?");
int pixelsHeight = fm.height();
```

Transformation

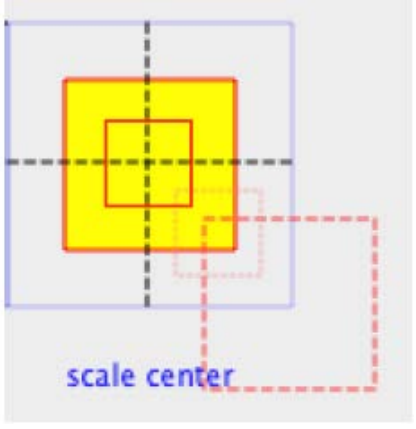
- Manipulating the coordinate system
  - translate(x,y)
  - scale(sx,sy)
  - rotate(a)
  - shear(sh,
- Demo



Transform
and Center

- scale(sx, sy)
  - scales around QPoint(0,0)
- Same applies to all transform operations
- Scale around center?

```
painter.drawRect(r);
painter.translate(r.center());
painter.scale(sx,sy);
painter.translate(-r.center());
// draw center-scaled rect
painter.drawRect(r);
```



QPainterPath

- Container for painting operations
- Enables reuse of shapes

```
QPainterPath path;
path.addRect(20, 20, 60, 60);
path.moveTo(0, 0);
path.cubicTo(99, 0, 50, 50, 99, 99);
path.cubicTo(0, 99, 50, 50, 0, 0);
painter.drawPath(path);
```

- Path information

  `controlPointRect()` - rect containing all points

  `contains()` - test if given shape is inside path

  `intersects()` - test given shape intersects path

- Demo

Hands-on

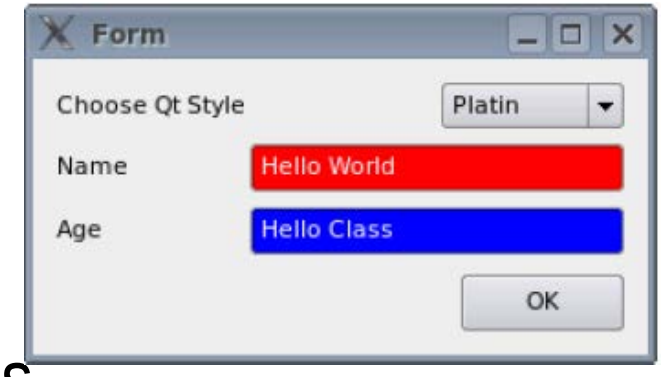
- Lab 7: Pie chart
  - Objectives
  - Template code

Painting & Styling

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Qt Style Sheets

- Mechanism to customize appearance of widgets
  - Additional to subclassing QStyle
- Inspired by HTML CSS
- Textual specifications of styles
- Applying Style Sheets
  - QApplication::setStyleSheet(sheet)
    - On whole application
  - QWidget::setStyleSheet(sheet)
    - On a specific widget (incl. child widgets)
- Demo



CSS Rules

CSS Rule

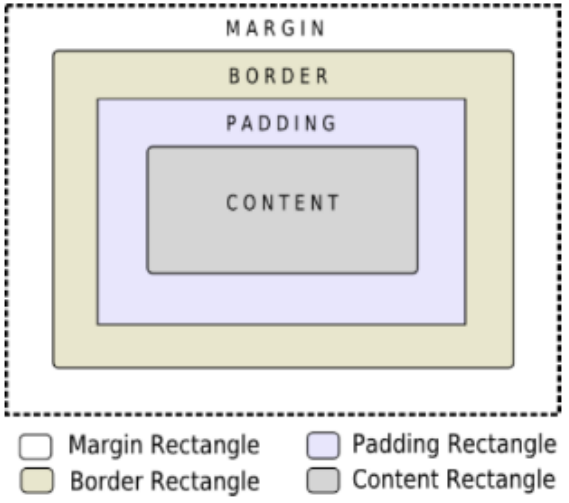
selector { property : value; property : value }

- Selector: specifies the widgets
- Property/value pairs: specify properties to change.

```
QPushButton {color:red; background-color:white}
```

- Examples of stylable elements
  - Colors, fonts, pen style, alignment.
  - Background images.
  - Position and size of sub controls.
  - Border and padding of the widget itself.
- Reference of stylable elements
  - stylesheet-reference

The Box Model



- Every widget treated as box
- Four concentric rectangles
  - Margin, Border, Padding, Content
- Customizing QPushButton

```
QPushButton {
border-width: 2px;
border-radius: 10px;
padding: 6px;
// ...
}
```

- By default, margin, border-width, and padding are 0

Selector Types

- *{ } // Universal selector
  - All widgets
- QPushButton { } // Type Selector
  - All instances of class
- .QPushButton { } // Class Selector
  - All instances of class, but not subclasses
- QPushButton#objectName // ID Selector
  - All Instances of class with objectName
- QDialog QPushButton { } // Descendant Selector
  - All instances of QPushButton which are child of QDialog
- QDialog > QPushButton { } // Direct Child Selector
  - All instances of QPushButton which are direct child of QDialog
- QPushButton[enabled="true"] // Property Selector
  - All instances of class which match property

Selector Details

- Property Selector
  - If property changes it is required to re-set style sheet
- Combining Selectors
  - `QLineEdit, QComboBox, QPushButton { color: red }`
- Pseudo-States
  - Restrict selector based on widget's state
  - Example: `QPushButton:hover {color:red}`
- Demo
- Selecting Subcontrols
  - Access subcontrols of complex widgets
    - as `QComboBox, QSpinBox`, ...
  - `QComboBox::drop-down { image: url(dropdown.png) }`
- Subcontrols positioned relative to other elements
  - Change using subcontrol-origin and subcontrol-position

Cascading

Effective style sheet obtained by merging

1. Widgets's ancestor (parent, grandparent, etc.)
2. Application stylesheet

- On conflict: widget own style sheet preferred

```
qApp->setStyleSheet("QPushButton { color: white }");
button->setStyleSheet("* { color: blue }");
```

- Style on button forces button to have blue text
  - In spite of more specific application rule
- Demo

Selector Specifity

- Conflict: When rules on same level specify same property
  - Specificity of selectors apply

```
QPushButton:hover { color: white }
QPushButton { color: red }
```

  - Selectors with pseudo-states are more specific
- Calculating selector's specificity
  - **a** Count number of ID attributes in selector
  - **b** Count number of property specifications
  - **c** Count number of class names
  - Concatenate numbers **a**-**b**-**c**. Highest score wins.
  - If rules scores equal, use last declared rule

```
QPushButton {} /* a=0 b=0 c=1 -> specificity = 1 */
QPushButton#ok {} /* a=1 b=0 c=1 -> specificity = 101
*/
```

- Demo

Hands-on

- Try this demo code and
  - Investigate style sheet
  - Modify style sheet
  - Remove style sheet and implement your own

Application creation

Objectives

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

Objectives

We will create an application to show fundamental concepts

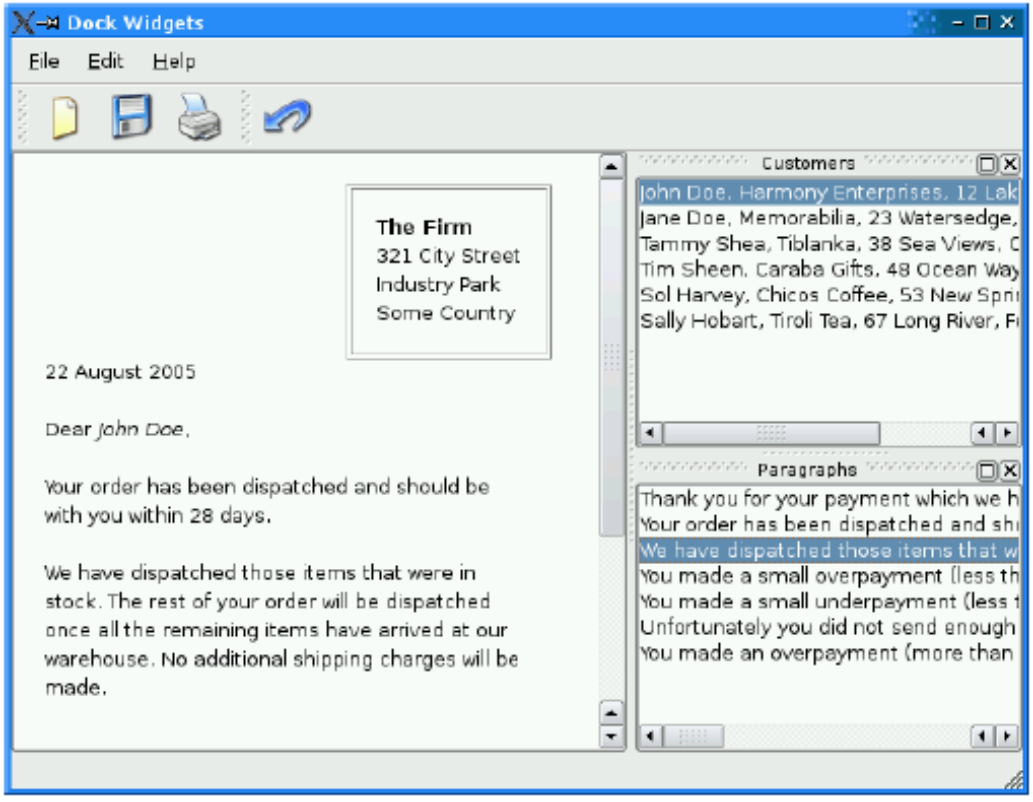
- **Main Window:** How a typical main window is structured
- **Settings:** Store/Restore application settings
- **Resources:** Adding icons and other files to your application
- **Deployment:** Distributing your application

Application creation

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

Application Ingredients

- Main window with
  - Menu bar
  - Tool bar, Status bar
  - Central widget
  - Often a dock window
- Settings (saving state)
- Resources (e.g icons)
- Translation
- Load/Save documents

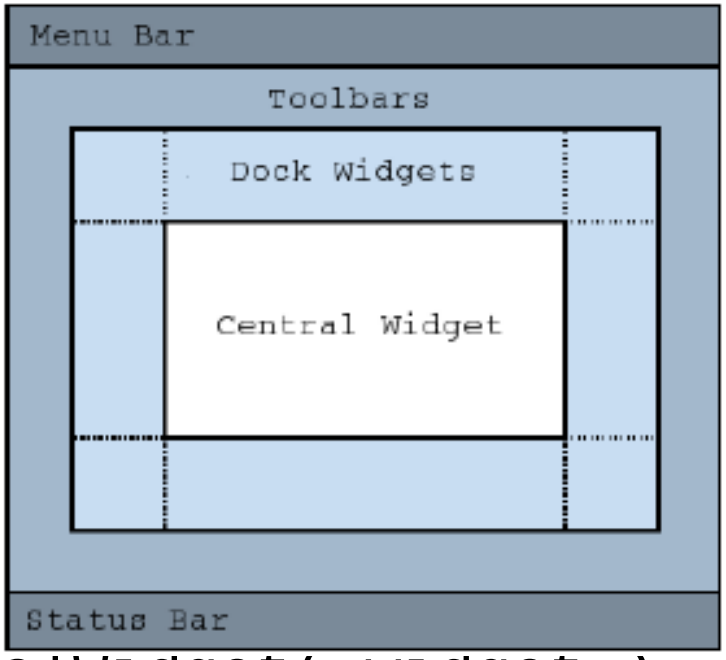


Not a complete list

ΣMERTXE

Main Window

- QMainWindow: main application window
- Has own layout
  - Central Widget
  - QMenuBar
  - QToolBar
  - QDockWidget
  - QStatusBar
- Central Widget
  - **`QMainWindow`**`::setCentralWidget( widget )`
  - Just any widget object



EMERTXE

QAction

Action is an abstract user interface command

- Emits signal triggered on execution
  - Connected slot performs action
- Added to menus, toolbar, key shortcuts
- Each performs same way
  - Regardless of user interface used

```
void MainWindow::setupActions() {
QAction* action = new QAction(tr("Open ..."), this);
action->setIcon(QIcon(":/images/open.png"));
action->setShortcut(QKeySequence::Open);
action->setStatusTip(tr("Open file"));
connect(action, SIGNAL(triggered()), this, SLOT(onOpen()));
menu->addAction(action);
toolbar->addAction(action);
```

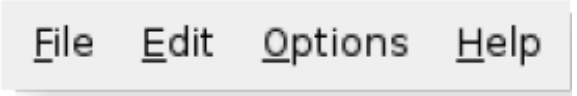
- Qaction Documentation

QAction capabilities

- setEnabled(bool)
  - Enables disables actions
  - In menu and toolbars, etc...
- setCheckable(bool)
  - Switches checkable state (on/off)
  - setChecked(bool) toggles checked state
- setData(QVariant)
  - Stores data with the action

Menu Bar

- **QMenuBar**: a horizontal menu bar
- **QMenu**: represents a menu
  - indicates action state
- **QAction**: menu items added to **QMenu**




```
void MainWindow::setupMenuBar() {
QMenuBar* bar = menuBar();
QMenu* menu = bar->addMenu(tr("&File"));
menu->addAction(action);
menu->addSeparator();
QMenu* subMenu = menu->addMenu(tr("Sub Menu"));
...
```

EMERTXE

QToolBar

- Movable panel ...
  - Contains set of controls
  - Can be horizontal or vertical
- QMainWindow::addToolbar(toolbar)
  - Adds toolbar to main window
- QMainWindow::addToolBarBreak()
  - Adds section splitter
- QToolBar::addAction(action)
  - Adds action to toolbar
- QToolBar::addWidget(widget)
  - Adds widget to toolbar

```
void MainWindow::setupToolBar() {
QToolBar* bar = addToolBar(tr("File"));
bar->addAction(action);
bar->addSeparator();
bar->addWidget(new QLineEdit(tr("Find ...")));
...
```

ΣMERTXE

QToolButton

- Quick-access button to commands or options
- Used when adding action to QToolBar
- Can be used instead QPushButton
  - Different visual appearance!
- Advantage: allows to attach action

```
QToolButton* button = new QToolButton(this);
button->setDefaultAction(action);
// Can have a menu
button->setMenu(menu);
// Shows menu indicator on button
button->setPopupMode(QToolButton::MenuButtonPopup);
// Control over text + icon placements
button-
>setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
```

QStatusBar

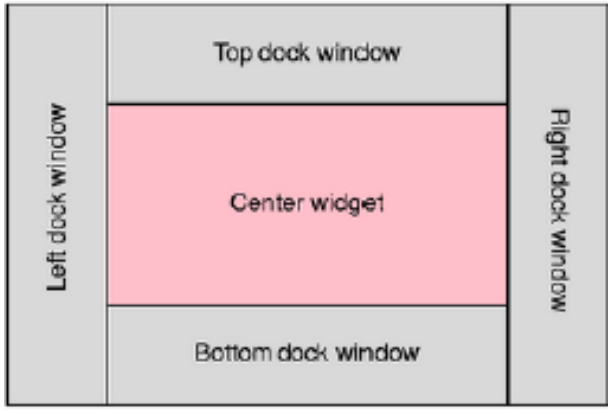
Horizontal bar
Suitable for presenting status information

- showMessage(message, timeout)
  - Displays temporary message for specified milli-seconds
- clearMessage()
  - Removes any temporary message
- addWidget() or addPermanentWidget()
  - Normal, permanent messages displayed by widget

```
void MainWindow::createStatusBar() {
QStatusBar* bar = statusBar();
bar->showMessage(tr("Ready"));
bar->addWidget(new QLabel(tr("Label on
StatusBar")));
```

QDockWidget

- Window docked into main window
- Qt::DockWidgetArea enum
  - Left, Right, Top, Bottom dock areas
- QMainWindow::setCorner(corner,area)
  - Sets area to occupy specified corner
- QMainWindow::setDockOptions(options)
  - Specifies docking behavior (animated, nested, tabbed, ...)




```
void MainWindow::createDockWidget() {
QDockWidget *dock = new QDockWidget(tr("Title"),
this);
dock->setAllowedAreas(Qt::LeftDockWidgetArea);
QListWidget *widget = new QListWidget(dock);
dock->setWidget(widget);
addDockWidget(Qt::LeftDockWidgetArea, dock);
```

QMenu
and Context Menus

- Launch via event handler

```
void MyWidget::contextMenuEvent(event) {
m_contextMenu->exec(event->globalPos());
```

- or signal customContextMenuRequested()
  - Connect to signal to show context menu
- Or via QWidget::actions() list
  - QWidget::addAction(action)
  - setContextMenuPolicy(Qt::ActionsContextMenu)
  - Displays QWidget::actions() as context menu

Hands-on

- Lab 8: Text editor
  - Objectives
  - Template code

Application creation

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

QSettings

- Configure QSettings

```
QCoreApplication::setOrganizationName("MyCompany");
QCoreApplication::setOrganizationDomain( "mycompany.com" );
QCoreApplication::setApplicationName("My Application");
```

- Typical usage

```
QSettings settings;
settings.setValue("group/value", 68);
int value = settings.value("group/value").toInt();
```

- Values are stored as QVariant
- Keys form hierarchies using '/'
  - or use beginGroup(prefix) / endGroup()
- value() excepts default value
  - settings.value("group/value", 68).toInt()
- If value not found and default not specified

  Invalid QVariant() returned

Restoring State

- Store geometry of application

```
void MainWindow::writeSettings() {
QSettings settings;
settings.setValue("MainWindow/size", size());
settings.setValue("MainWindow/pos", pos());
}
```

- Restore geometry of application

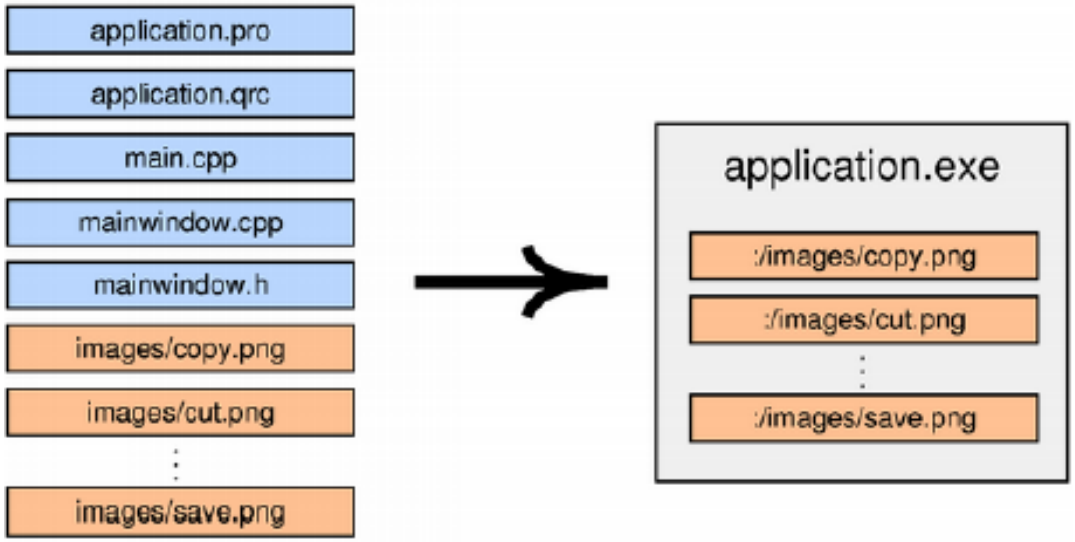
```
void MainWindow::readSettings() {
QSettings settings;
settings.beginGroup("MainWindow");
resize(settings.value("size", QSize(400,
400)).toSize());
move(settings.value("pos", QPoint(200,
200)).toPoint());
settings.endGroup();
}
```

Application creation

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

Resource System

- Platform-independent mechanism for storing binary files
  - Not limited to images
- Resource files stored in application's executable
- Useful if application requires files
  - E.g. icons, translation files, sounds
  - Don't risk of losing files, easier deployment
- Docs



Using Resources

- Resources specified in .qrc file

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
<file>images/copy.png</file>
<file>images/cut.png</file>
</qresource>
</RCC>
```

  - Can be created using QtCreator
- Resources are accessible with ':' prefix
  - Example: ":/images/cut.png"
  - Simply use resource path instead of file name
  - QIcon(":/images/cut.png")
- To compile resource, edit .pro file
  - RESOURCES += application.qrc
  - qmake produces make rules to generate binary file

Hands-On

- Use your previous text editor, to use Qt resource system for icons
- Tip: You can use Qt Creator to create QRC files

Application creation

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

Ways of Deploying

- Static Linking
  - Results in stand-alone executable
  - +Only few files to deploy
  - -Executables are large
  - -No flexibility
  - -You cannot deploy plugins
- Shared Libraries
  - +Can deploy plugins
  - +Qt libs shared between applications
  - +Smaller, more flexible executables
  - -More files to deploy
- Qt is by default compiled as shared library
- If Qt is pre-installed on system
  - Use shared libraries approach
- Reference Documentation

Deployment

- Shared Library Version
  - If Qt is not a system library
    - Need to redistribute Qt libs with application
  - Minimal deployment
    - Libraries used by application
    - Plugins used by Qt
  - Ensure Qt libraries use correct path to find Qt plugins
- Static Linkage Version
  - Build Qt statically
    - $QTDIR/configure -static <your other options>
    - Specify required options (e.g. sql drivers)
  - Link application against Qt
  - Check that application runs stand-alone
    - Copy application to machine without Qt and run it

Dialogs module

Dialogs
and Designer

- Dialogs
- Common Dialogs
- Qt Designer

Dialogs
and Designer

- **Custom Dialogs**
  - Modality
  - Inheriting QDialog
  - Dialog buttons
- **Predefined Dialogs**
  - File, color, input and font dialogs
  - Message boxes
  - Progress dialogs
  - Wizard dialogs
- **Qt Designer**
  - Design UI Forms
  - Using forms in your code
  - Dynamic form loading

Dialogs
and Designer

- Dialogs
- Common Dialogs
- Qt Designer

QDialog

- Base class of dialog window widgets
- General Dialogs can have 2 modes
- Modal dialog
  - Remains in foreground, until closed
  - Blocks input to remaining application
  - Example: Configuration dialog
- Modeless dialog
  - Operates independently in application
  - Example: Find/Search dialog
- Modal dialog example

```
MyDialog dialog(this);
dialog.setMyInput(text);
if(dialog.exec() == Dialog::Accepted) {
// exec blocks until user closes dialog
```

Modeless Dialog

- Use show()
  - Displays dialog
  - Returns control to caller

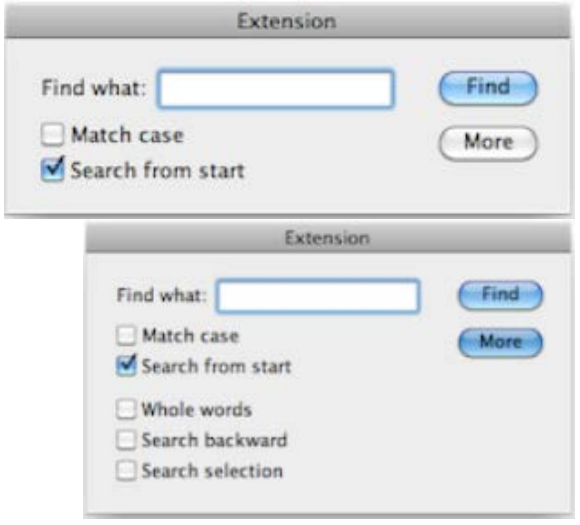
```
void EditorWindow::find() {
if (!m_findDialog) {
m_findDialog = new FindDialog(this);
connect(m_findDialog, SIGNAL(findNext()),
this, SLOT(onFindNext()));
}
m_findDialog->show(); // returns immediately
m_findDialog->raise(); // on top of other windows
m_findDialog->activateWindow(); // keyboard focus
}
```

Custom Dialogs

- Inherit from QDialog
- Create and layout widgets
- Use QDialogButtonBox for dialog buttons
  - Connect buttons to accept()/reject()
- Override accept()/reject()

```
MyDialog::MyDialog(QWidget *parent) : QDialog(parent) {
m_label = new QLabel(tr("Input Text"), this);
m_edit = new QLineEdit(this);
m_box = new QDialogButtonBox( QDialogButtonBox::Ok|
QDialogButtonBox::Cancel, this);
connect(m_box, SIGNAL(accepted()), this, SLOT(accept()));
connect(m_box, SIGNAL(rejected()), this, SLOT(reject()));
... // layout widgets
}
void MyDialog::accept() { // customize close behaviour
if(isDataValid()) { QDialog::accept() }
}
```

Deletion and Extension



- Deletion of dialogs
  - No need to keep dialogs around forever
  - Call QObject::deleteLater()
  - Or setAttribute(Qt::WA_DeleteOnClose)
  - Or override closeEvent()
- Dialogs with extensions:
  - QWidget::show()/hide() used on extension

```
m_more = new QPushButton(tr("&More"));
m_more->setCheckable(true);
m_extension = new QWidget(this);
// add your widgets to extension
m_extension->hide();
connect(m_more, SIGNAL(toggled(bool)),
m_extension, SLOT(setVisible(bool)));
```

- Example

Dialogs
and Designer

- Dialogs
- Common Dialogs
- Qt Designer

QFileDialog

- Allow users to select files or directories
- Asking for a file name

```
QString fileName =
QFileDialog::getOpenFileName(this, tr("Open File"));
if(!fileName.isNull()) {
// do something useful
}
```

- QFileDialog::getOpenFileNames()
  - Returns one or more selected existing files
- QFileDialog::getSaveFileName()
  - Returns a file name. File does not have to exist.
- QFileDialog::getExistingDirectory()
  - Returns an existing directory.
- setFilter("Image Files (*.png *.jpg *.bmp)")
  - Displays files matching the patterns

QMessageBox

- Provides a modal dialog for ...
  - informing the user
  - asking a question and receiving an answer
- Typical usage, questioning a user

```
QMessageBox::StandardButton ret =
QMessageBox::question(parent, title, text);
if(ret == QMessageBox::Ok) {
// do something useful
}
```

- Very flexible in appearance
  - Reference documentation
- Other convenience methods
  - `QMessageBox::information(...)`
  - `QMessageBox::warning(...)`
  - `QMessageBox::critical(...)`
  - `QMessageBox::about(...)`

QProgressDialog

- Provides feedback on the progress of a slow operation

```
QProgressDialog dialog("Copy", "Abort", 0, count, this);
dialog.setWindowModality(Qt::WindowModal);
for (int i = 0; i < count; i++) {
dialog.setValue(i);
if (dialog.wasCanceled()) { break; }
//... copy one file
}
dialog.setValue(count); // ensure set to maximum
```

- Initialize with setValue(0)
  - Otherwise estimation of duration will not work
- When operation progresses, check for cancel
  - QProgressDialog::wasCanceled()
  - Or connect to QProgressDialog::canceled()
- To stay reactive call QApplication::processEvents()
- See Documentation

QErrorMessage

- Similar to QMessageBox with checkbox
- Asks if message shall be displayed again

```
m_error = new QErrorMessage(this);
m_error->showMessage(message, type);
```

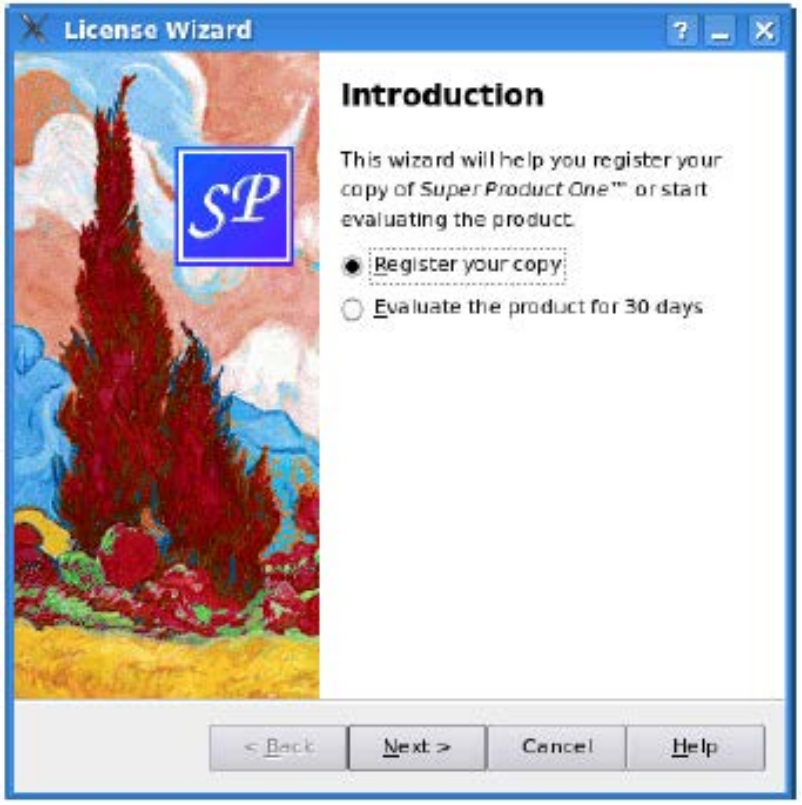
- Messages will be queued
- QErrorMessage::qtHandler()
  - installs an error handler for debugging
  - Shows qDebug(), qWarning() and qFatal() messages in QErrorMessage box

Other Common Dialogs

- Asking for Input - QInputDialog
  - QInputDialog::getText(...)
  - QInputDialog::getInt(...)
  - QInputDialog::getDouble(...)
  - QInputDialog::getItem(...)
- Selecting Color - QColorDialog
  - QColorDialog::getColor(...)
- Selecting Font - QFontDialog
  - QFontDialog::getFont(...)
- Example

Qwizard
Guiding the user

- Input dialog
  - Consisting of sequence of pages
- Purpose: Guide user through process
  - Page by page
- Supports
  - Linear and non-linear wizards
  - Registering and using fields
  - Access to pages by ID
  - Page initialization and cleanup
  - Title, sub-title
  - Logo, banner, watermark, background
  - Documentation
- Each page is a QWizardPage
- QWizard::addPage()
  - Adds page to wizard
- example



ΣMERTXE

Hands-on

- Lab 9: Dialog
  - Objectives
  - Template code

Summary

- When would you use a modal dialog, and when would you use a non-modal dialog?
- When should you call exec() and when should you call show()?
- Can you bring up a modal dialog, when a modal dialog is already active?
- When do you need to keep widgets as instance variables?
- What is the problem with this code:

```
QDialog *dialog = new QDialog(parent);
QCheckBox *box = new QCheckBox(dialog);
```

EMERTXE

THANK YOU